Pattern Recognition Letters
journal homepage: www.elsevier.com

Optimising the Volgenant-Jonker Algorithm for Approximating Graph Edit Distance

William Jones[a], Aziem Chawdhary[a], Andy King[a]

[a]*School of Computing, University of Kent, Canterbury, CT2 7NF, United Kingdom*

ABSTRACT

Although it is agreed that the Volgenant-Jonker (VJ) algorithm provides a fast way to approximate graph edit distance (GED), until now nobody has reported how the VJ algorithm can be tuned for this task. To this end, we revisit VJ and propose a series of refinements that improve both the speed and memory footprint without sacrificing accuracy in the GED approximation. We quantify the effectiveness of these optimisations by measuring distortion between control-flow graphs: a problem that arises in malware matching. We also document an unexpected behavioural property of VJ in which the time required to find shortest paths to unassigned vertices decreases as graph size increases, and explain how this phenomenon relates to the birthday paradox.

1. Introduction

Attributed graphs have risen in popularity in various subfields of pattern recognition (Conte et al., 2004; Livi and Rizzi, 2013), but one fundamental problem of attributed graphs remains: the problem of how to quantify the similarity of two such graphs (Bunke and Allermann, 1983). One promising class of methods is based on the measure of Graph Edit Distance (GED), (Gao et al., 2010; Sanfeliu and Fu, 1983) which measures the similarity of two graphs as the minimum number of edit operations needed to convert one graph to another. More precisely, suppose $G = \langle V, E, \ell \rangle$ is an attributed directed graph where $E \subseteq V \times V$ and $\ell : V \to \Sigma$ assigns each vertex to an attribute (or label) drawn from an alphabet Σ. (In the general case, edges can also be similarly attributed.) An edit operation on a graph G_1 inserts or deletes an individual vertex, inserts or deletes an edge, or reassigns a vertex to an attribute, to obtain a new graph G_2. Applying a sequence of $n-1$ edit operations gives a sequence of n graphs $G_1, G_2, \ldots, G_n$. Since the cost of edit operations is not necessarily uniform, in the more general form, each edit operation has an associated edit cost as defined by a cost function. The GED between two graphs G and G' is the minimum sum of edit operation costs. GED has proven to be useful (Bourquin et al., 2013; Conte et al., 2004; Myers et al., 2000; Riesen and Bunke, 2009; Eshera and Fu, 1984) because it is an error tolerant measure of similarity.

However, computing GED is equivalent to finding an optimal permutation matrix (Zeng et al., 2009), an NP-hard problem. Fast but suboptimal approaches have thus risen to prominence (Riesen and Bunke, 2009), in which GED is approximated by solving a linear sum assignment problem. Of those algorithms proposed for solving this problem, the Volgenant-Jonker (VJ) algorithm (Jonker and Volgenant, 1987) is the most efficient. This paper takes VJ as the starting point, and explores how it can be improved for the specific task of GED computation. Others (Serratosa, 2014; Serratosa and Cortés, 2014), recognising the need to speed up bipartite graph matching, have redefined the concept of edit distance to make it more amenable to calculation. Our approach differs from these works because we exploit the highly regular structure of the cost matrix and the redundancy that this implies for the VJ algorithm, instead of using an alternative definition of edit distance. This paper extends (Jones et al., 2015) by fully explaining how the fundamental operations of the VJ algorithm can be improved to take advantage of a specific structure of the cost matrix. In all, our paper makes the following contributions:

1. It gives a more detailed description of our workshop paper Jones et al. (2015);
2. It shows how the VJ algorithm can be tuned to GED computation; It also elucidates details of VJ itself;
3. It quantifies the ensuing speedup and decrease in memory requirements, demonstrating the efficacy both on randomly generated data and in real world applications;
4. It shows how the speedup can be explained with respect to cache usage;
5. It reports and discusses an emergent behaviour in which the time taken on the shortest path calculations decreases

as the problem size increases;

The exposition of the paper is structured as follows: To keep the paper self-contained, section 2, explains how GED is related to the linear sum assignment problem, and section 3 describes the classical VJ algorithm. Section 4 introduces the proposed optimisations, and Section 5 presents the experimental results, comparing the improved algorithm with the original. Section 6 discusses how the relation between the VJ and other proposals for improving bipartite graph matching. Section 7 concludes.

2. The Linear Assignment Problem and GED

The linear assignment problem is one of fundamental problems of combinatorial optimisation, that is, the problem of assigning n unique agents to n unique tasks such that the total summed cost of these assignments is minimal. More exactly, given two sets $|A| = n = |B|$ and an associated $n \times n$ cost matrix C in which C_{ij} corresponds to the cost of assigning the i-th element of A to the j-th element of B. Then the linear assignment problem is that of finding a bijection $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ which assigns a column $f(i)$ to each row i in a way that minimises $\sum_{i=1}^{n} C_{i,f(i)}$. Identically, we can also view the problem as finding a permutation $p_1, p_2, \ldots, p_n$ on $1, 2, \ldots, n$ that minimises $\sum_{i=0}^{n} C_{i,p_i}$ (Munkres, 1957; Burkard and Cela, 1999).

Any algorithm capable of finding an optimal solution to the assignment problem, such as the Hungarian algorithm (Kuhn, 1955), can be used in the derivation of a (suboptimal) solution to the GED problem (Riesen et al., 2007). Consider graphs defined similar to before. When $|V_1| = n = |V_2|$ the GED between $G_1 = \langle V_1, E_1, \ell_1 \rangle$ and $G_2 = \langle V_2, E_2, \ell_2 \rangle$ can be approximated by solving a linear assignment problem using an $n \times n$ matrix C where $C_{i,j}$ denotes the cost of substituting vertex i for vertex j. However, an optimal minimal solution to the assignment problem does not guarantee an optimal solution to the GED problem; every vertex operation described by this solution also entails edge operations that are not accounted for in this calculation. Another limitation of this particular model is that it can only be easily applied when $|V_1| = |V_2|$. A more general approach (Riesen and Bunke, 2009) addresses both these problems by working on an extended cost matrix defined as follows:

$$C = \left[\begin{array}{cccc|cccc} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} & \infty & \cdots & \infty \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & \infty & c_{2,\epsilon} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \infty \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & \infty & \cdots & \infty & c_{n,\epsilon} \\ \hline c_{\epsilon,1} & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \infty & c_{\epsilon,2} & \cdots & \infty & 0 & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\ \infty & \cdots & \infty & c_{\epsilon,m} & 0 & \cdots & 0 & 0 \end{array}\right]$$

where $n = |V_1|$ and $m = |V_2|$. The top left quadrant of $c_{i,j}$ describes the cost of substituting vertex i with vertex j. The top right hand corner $c_{i,\varepsilon}$ the cost of deleting vertex i. The bottom left quadrant $c_{\varepsilon,j}$ denotes the cost of inserting vertex j.

It is convenient to augment the set of attributes Σ, henceforth referred to as the set of real attributes, with a virtual attribute, denoted ϵ. The virtual attribute is introduced to mark those vertices that do not participate in the graphs. These are called the virtual vertices; all other vertices are considered real. Deletion is then assigning a virtual attribute to a real vertex. Conversely, insertion is assigning a real attribute to a virtual vertex. Substitution is seen as simply assigning a real attribute to a real vertex. However, in order to complete the cost matrix, we must quantify the cost of assigning a virtual vertex to the virtual attribute, an action that is vacuous. This naturally has a cost of 0, thus the bottom right quadrant that completes the matrix is uniformly 0.

Note that even with this extension (Riesen and Bunke, 2009) the cost matrix only considers vertex information. A further refinement that uses local edge information to generate a more accurate approximation to GED can be defined as follows: For a given graph G_k we use E_k and ℓ_k to compute a lower bound on GED by considering attributes on the incoming neighbours of a given vertex. To be precise, let

$$In_k(j) = \{\ell_k(i) \mid \langle i, j \rangle \in E_k\}$$

be the set of attributes on the incoming neighbours of a given vertex j. For a given vertex i in G_1 and a given vertex j in G_2, let S_1 be those attributes on the incoming neighbours of i which are not assigned to incoming neighbours of j. S_2 is defined conversely, so that:

$$S_1 = \{\sigma \in In_1(i) \mid \sigma \notin In_2(j)\}$$
$$S_2 = \{\sigma \in In_2(j) \mid \sigma \notin In_1(i)\}$$

The maximal cardinality of S_1 and S_2 then quantifies how many attributes the incoming neighbours vertices i and j differ by:

$$e_{i,j} = \max(|S_1|, |S_2|)$$

The cost matrix is then defined $c_{i,j} = d_{i,j} + e_{i,j}$ where

$$d_{i,j} = \begin{cases} 1 & \text{if } \ell_1(i) \neq \ell_2(j) \\ 0 & \text{otherwise} \end{cases}$$

Then the $d_{i,j}$ component of $c_{i,j}$ accounts for any difference in attribution between vertex i and vertex j, and the $e_{i,j}$ component accounts for reassigning their incoming neighbours. The diagonals $c_{i,\epsilon}$ and $c_{\epsilon,i}$ are degenerative and defined as above with $In_k(\epsilon) = \emptyset$.

3. The classical VJ algorithm

To support the reader, we provide the intuition behind the algorithmic details given in Jonker and Volgenant (1987). By way of support, we first overview the algorithm at the macroscopic scale, before drilling into the microscopic detail in subsequent sections. This overview is designed as a support text and commentary on the original seminal work.

3.1. Overview

The VJ algorithm (Jonker and Volgenant, 1987) is a shortest path algorithm solved via a dual method. The algorithm consists of two main steps, which are outlined in the sections that follow:

$$\begin{bmatrix} 1 & 7 & 6 \\ 5 & 2 & 3 \\ 8 & 9 & 4 \end{bmatrix} \quad \begin{bmatrix} 0 & 5 & 3 \\ 4 & 0 & 0 \\ 7 & 7 & 1 \end{bmatrix} \quad \begin{bmatrix} 3 & 5 & 3 \\ 7 & 0 & 0 \\ 10 & 7 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 2 & 0 \\ 7 & 0 & 0 \\ 10 & 7 & 1 \end{bmatrix}$$
$$(a) \qquad (b) \qquad (c) \qquad (d)$$

Fig. 1. (a) Example cost matrix; (b) After column reduction; (c) After anti-column reduction; (d) After row reduction (reduction transfer)

1. Initialisation in three stages: (a) column reduction; (b) reduction transfer; and (c) augmenting row reduction. These are somewhat akin to the initialisation steps of the Hungarian algorithm (Kuhn, 1955) and serve a similar purpose: to quickly make an initial set of tentative assignments using a fast, but potentially incomplete method. These assignments correspond to a bijection f from a subset of the row indices to a subset of the column indices. In addition to this, the assignments made in initialisation make changes to the entries in the cost matrix. This further speeds up augmentation.
2. The heart of the algorithm is augmentation which incrementally constructs the bijective map f. Each iteration of augmentation adds one element (a row index) to the domain of f and another element (a column index) to codomain of f. Augmentation is repeatedly applied until f is complete. Each step of augmentation finds an alternating path that starts with a row index outside the domain of f and ends with a column index outside the codomain of f. An alternating path is a sequence of row and column indice pairs where the internal rows and columns in the sequence appear in the domain and codomain of f. Each row and column pair is associated with a cost, by virtue of the cost matrix, and the alternating path is calculated to minimise the cumulative cost. Each alternating path that is found acts to increase the size of the domain and codomain, which is why termination is ultimately assured.

3.2. Initialisation

Column reduction. The first step of initialisation is column reduction, in which a positive value is subtracted from each element of a column. Starting at the last column, an optimisation suggested by Jonker and Volgenant (1987), the VJ algorithm reduces each column by its minimum element such that each column contains a zero. Figure 1(b) illustrates the result of column reduction. As the matrix is scanned right-to-left, each column is assigned, whenever possible, to a unique row that contains a zero in that column. Column 3 is assigned to row 2 (and vice versa), and column 1 is assigned to row 1 (and vice versa), but column 2 will remain unassigned (as does row 3).

Reduction transfer. The second step of initialisation is reduction transfer, which is applied to further reduce assigned rows and in which a positive value is subtracted from each element of a row. Since any assigned row will, by necessity, contain at least one zero, row reduction cannot be applied, without introducing a negative entry (because column reduction has taken place). Consider for illustration row 1 of Figure 1(b). Thus a form of inverse column reduction is applied to the column corresponding to the minimum of row 1, which is column 1, to give the matrix depicted in Figure 1(c). Row reduction is then applied, to decrease row 1 by the least entry greater or equal to the current minimum (henceforth called the second minimum), which in the case of Figure 1(c) is the value 3. The result of this row reduction is illustrated in Figure 1(d), albeit at the expense of reduction in the selected column. This exchange in reduction value, by the second minimum 3, between column 1 and row 1, is called reduction transfer. Aside from the benefits in reducing assigned rows, this step is particularly beneficial for expediting augmentation. By decreasing the reduction in the assigned column, the entries in this column are more likely to higher than the entries in an unassigned column, making connections to unassigned columns more favourable.

Augmenting row reduction. In the third phase of initialisation, an attempt is made to find a set of alternating paths, where each path, recall, starts in an unassigned row and ends in an unassigned column. For a given unassigned row i, VJ finds a column j_1 that contains the minimum entry e_1 and another column j_2 that contains the least entry e_2 such that $e_2 \geq e_1$. Row i is then reduced by e_2. If $e_2 > e_1$ then this incurs a negative value in column j_1, in which case, inverse column reduction is applied to column j_1 to eliminate the negative entry. Row i is then assigned to column j_1 regardless of whether this column is already assigned or not. If j_1 was previously assigned to a row k, then row k becomes unassigned and the procedure continues from row k. This repeats until either row k is matched to an unassigned column, or it becomes impossible to transfer reduction to the selected row k. Observe that reduction transfer provides a vehicle for constructing a path that alternates between rows and columns, hence the name alternating path.

3.3. Augmentation

For each unassigned row, the augmentation phase finds a shortest alternating path to an unassigned column. VJ modifies Dijkstra's algorithm (Dijkstra, 1959) to search for these shortest paths, where the notion of distance between a row and a column is the entry in the cost matrix.

The search starts at an unassigned row, say row i, and a shortest path is found from row i to a column j, and column j is added to the path. If column j was previously assigned to row k, then row k is also added to the path (though no changes are made until a complete path to an unassigned column is found). The distances from the row i to any given column are updated and replaced if it is possible to reach this column in a shorter distance via row k. Unlike classical Dijkstra, the search continues in this fashion until an unassigned column is found. After augmentation, the assignments to the cost matrix are updated so that all assignments in the current solution correspond to minimum entries in each row of the cost matrix.

Because augmentation uses Dijkstra's algorithm, VJ implicitly assumes that all costs are non-negative.

4. The Improved VJ Algorithm

This section proposes a number of improvements to the classical VJ algorithm, all of which follow from the regular structure of the GED cost matrix.

4.1. Representation

Given two graphs G_1 and G_2 with V_1 and V_2 vertices respectively, by far the largest data structure in the VJ algorithm is the cost matrix requiring the storage of $(n + m)^2$ entries where $n = |V_1|$ and $m = |V_2|$. However, nearly three quarters of these entries are fixed at either zero or infinity, as is illustrated in Figure 2(a). There are two natural ways to represent the data more densely without loss of information, namely, a row-by-row representation depicted in Figure 2(b) and a column-by-column representation illustrated in Figure 2(c). Both representations avoid storing the infinite values in the top-right and bottom-left quadrant of the cost matrix. The top half of the row-by-row representation can be combined with the left half of the column-by-column representation to obtain a denser representation still, which avoids explicitly storing the zeros in the bottom right quadrant of the cost matrix. The combination is illustrated in Figure 2(d); the net effect is that the cost matrix is represented with only $(n + 1) \times (m + 1) - 1$ entries.

Although row equality can no longer be checked by comparing one entry against another one-by-one, we argue that the representation does not impede tractability but, in fact, that the converse is true. Moreover, this matrix representation homogenises the column-by-column and row-by-row representations, which means that it simultaneously benefits both row- and column-based operations. It should be noted that the VJ algorithm cannot be directly applied to the new representation; as it stands VJ will not calculate a correct set of assignments to the original problem because we are not explicitly storing information relating to the infinite and zero entries.

4.2. Column Reduction

The new representation simplifies column reduction in two ways: First, almost half of the costs in the matrix are infinite and so will never be chosen as a minimum in a column. Second, nearly a quarter of the costs will be zero, and these zeros dictate that the column minimum will be zero. Hence only the position of the minimum need be computed in column reduction (rather than its position and value). This is because, while we can be assured that there will be no minimum value smaller than zero, it is entirely possible for one of our variable entries to be zero.

To take advantage of this, the matrix is considered as two separate blocks with different operations provided for each, as illustrated in Figure 3a. The leftmost entries are handled as before because, barring the loss of infinite values which are irrelevant to this operation, the old and new data-structures coincide. The rightmost entries that are stored in a single column, see Figure 2(c), correspond to the diagonal of the top-right quadrant. Recall that we reduce each column by the minimum in that column. Then these rightmost entries are only ever compared to infinite values and zero values (because they share a column with the null quadrant). Hence the reduction value will always be zero and only the position of the zero need be found. This can be further simplified in the case that the variable entries in the cost matrix are uniformly non-zero because this removes the need for position computation too. This effectively renders the result of column reduction to be predeterminable for any given cost matrix.

4.3. Reduction Transfer

Reduction transfer can be simplified in an analogous manner. As before, we have no need to search for minimum or second minimum values for column entries containing infinite values. We can operate a similar algorithmic split to the one used in column reduction, but since reduction transfer operates row by row instead of column by column, we use a top-to-bottom division instead, as depicted in Figure 3b. Similarly to column reduction, any row in the bottom half of the matrix will intersect the null quadrant, and will have a known minimum and second minimum values (both of which will be zero). At this point, we can no longer assume, as we did in column reduction, that some of our values are non-zero. This is because column reduction has taken place and costs in the matrix may have changed. Therefore we cannot, as before, eliminate the need to search for the position of these minimums, though it is possible to greatly speed up the search as we only need to check whether our variable entry in this row is non-zero.

4.4. Augmenting Row Reduction

Augmenting row reduction can be optimised in a similar fashion to the previous two steps. Once again, for each row being considered there is no point considering infinite costs as candidates for the minimum. Equally, when operating on the bottom half of the cost matrix, we can assume that the minima in a given row are zero. We can make this assumption because even after reduction transfer has potentially changed the reduction values of these entries, edit operations from the virtual label to the virtual label should always have zero cost. (Note that we could have made this assumption for reduction transfer too, but chose not employ this refinement as it introduces overhead and reduction transfer makes up only an extremely small fraction of total runtime.) This means that we only need to check if the non-fixed entries in a given row have been reduced to zero.

4.5. Augmentation

Augmentation can be improved due to the new cost matrix, but cannot be improved to the same degree as the steps of initialisation. This is because we can never eliminate the overarching need to iterate over each column for any given row because, although we should never make a direct path to a given column via an infinite entry, it is entirely possible that we might update the cost of such a column by finding an indirect path to it. Despite this, we can make several improvements which follow from the observation that while we can never rule out an indirect path to such a column, we can certainly rule out a direct

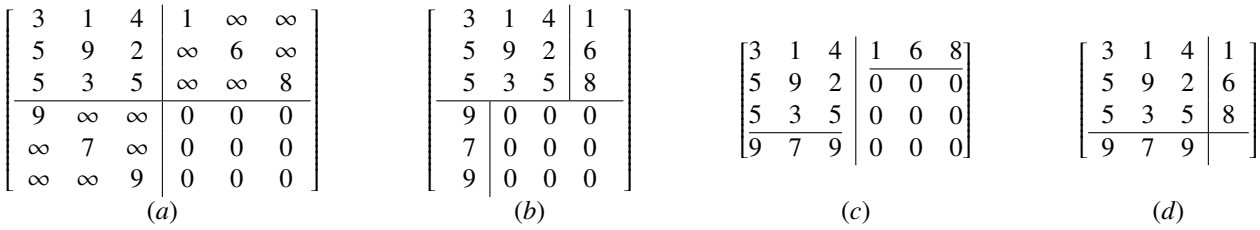

Fig. 2. (a) Original cost matrix; (b) Row-by-row representation with zeroes; (c) Column-by-column representation with zeroes; (d) Without zeroes.

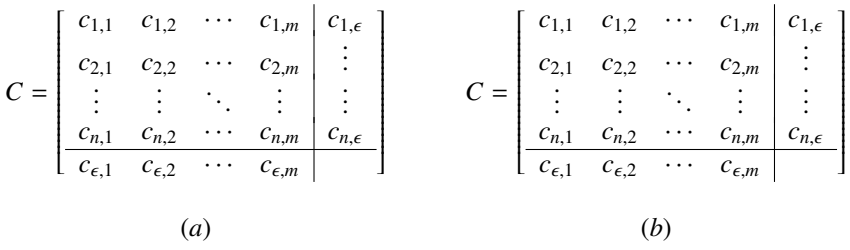

Fig. 3. (a) The split between the left- and right-hand block (the left is highlighted). (b) The between the top and bottom blocks (the top is highlighted).

path. Equally, it is possible to replace or skip a large number of matrix lookups of what would be fixed values in the cost matrix.

Unlike the previous steps, where the refinements reduce the complexity from $O(\max(n,m)^2)$ to $O(nm)$, the complexity of augmentation remains the same at $O((n+m)^3)$, since it is performing up to $n+m$ iterations of Dijkstra's algorithm which resides in $O((n+m)^2)$. As this dominates the complexity of the whole algorithm, the improvements do not place the algorithm in a new complexity class. Nevertheless, since the initialisation steps that we improved generally take at least half of the total runtime is our tests of the algorithm, the proposed improvements are truly worthwhile, showing a significant overall speedup. Furthermore, they do not come at the expense of increased memory consumption rather, work in tandem with the representation that reduces the memory footprint.

5. Experimental Results

To empirically assess the proposed improvements to VJ, two versions were implemented: the original version (VJ-ORG) and an improved version (VJ-IMP). Both were compared against a version developed by Jonker himself (VJ-CTRL), which was used as a control. All three versions were implemented in C++, applying consistent coding patterns throughout.

5.1. Evaluation on Random Data

Initially random square ($n = m$) cost matrices were used to provide a large corpus of data for comparing all versions of VJ. The improvements have least effect on square matrices and thus, if anything, the setup is biased against VJ-IMP. Costs and matrix sizes were chosen to approximate what might typically be encountered in malware matching. In this context a control-flow graph (CFG) is extracted from a binary for comparison against a database of CFGs derived from malware. BinSlayer (Bourquin et al., 2013) was used to derive CFGs from several medium-sized binaries. These possessed between 465 to 6984 vertices (basic blocks), and produced matrices where the costs rarely exceeded 2000 and never exceeded 3000. To cover a range of scenarios, matrices were populated with random values from cost ranges varying over 1-500 to 1-3000. Matrix sizes were also varied between 1000 and 14000, again to simulate CFGs. We also tested the resilience of the improvements over some much larger cost ranges (1-50000 and 1-10^9) for the full range of matrix sizes.

Figure 4 gives a series of plot lines, one for each cost range. Each data point on a line corresponds to the runtimes for a different matrix size, though the horizontal axis is normalised by the runtime for solving a 14000×14000 matrix. The vertical axis is normalised relative to the unimproved times, so that the instantaneous gradient quantifies the improvement over a range of costs and matrix sizes. Note that VJ-IMP is uniformly faster than both VJ-ORG and the third party implementation by Jonker, VJ-CTRL. We find the improved version of algorithm to be about twice as fast; the difference being more striking at lower costs and higher sizes. Although not represented by this data, there is only a very small performance advantage to VJ-ORG over VJ-CTRL which suggests that our implementations are consistent and our findings are robust. All three versions also produce the same output given the same input.

5.2. Evaluation on CFGs

The previous subsection showed how to speed-up the VJ algorithm. In this section we show that the changes to the VJ algorithm also speed-up graph edit distance computation. Figure 6 summarises some CFG comparisons for four different binaries, where the CFGs were derived using BinSlayer. Comparing $CFG_1 = \langle V_1, E_1, \ell_1 \rangle$ against $CFG_2 = \langle V_2, E_2, \ell_2 \rangle$ does not necessarily take the same time as comparing CFG_2 against CFG_1. This is because if $|V_1| < |V_2|$ then cost matrix will have a

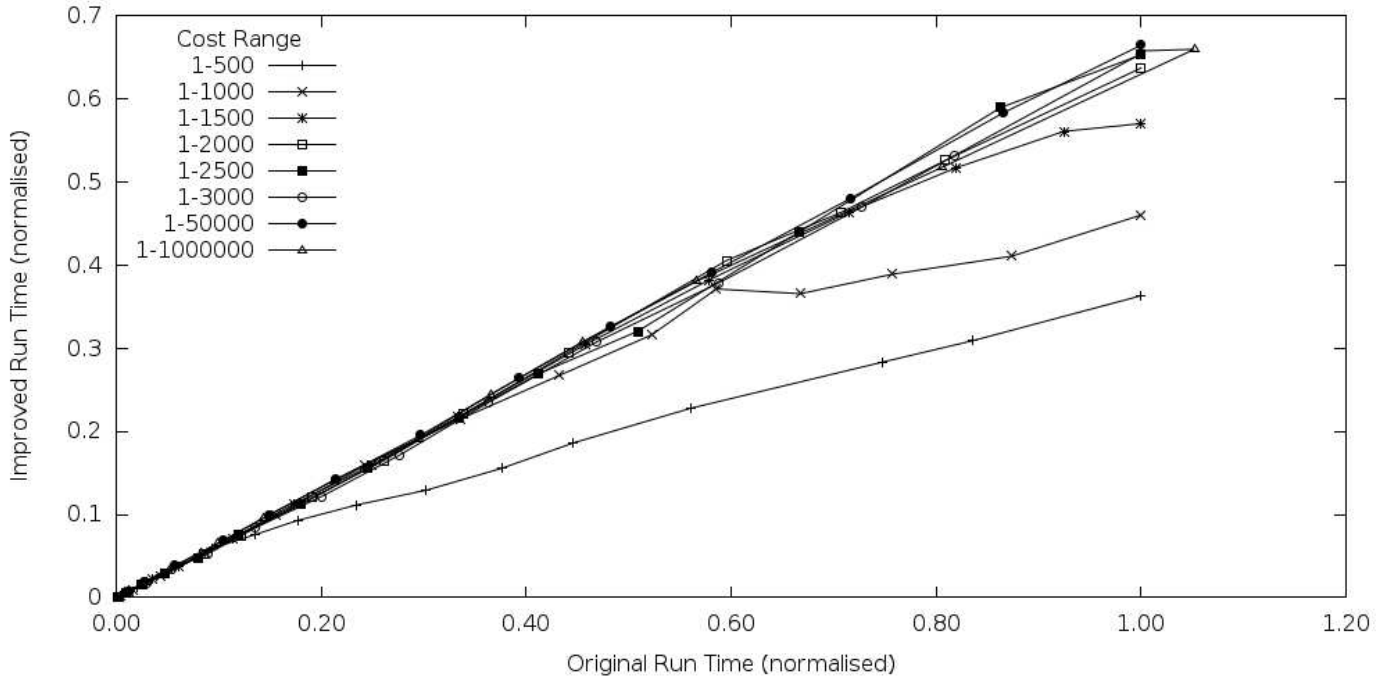

Fig. 4. VJ-IMP **against of the fastest of** VJ-ORG **and** VJ-CTRL **for various cost ranges and matrix sizes**

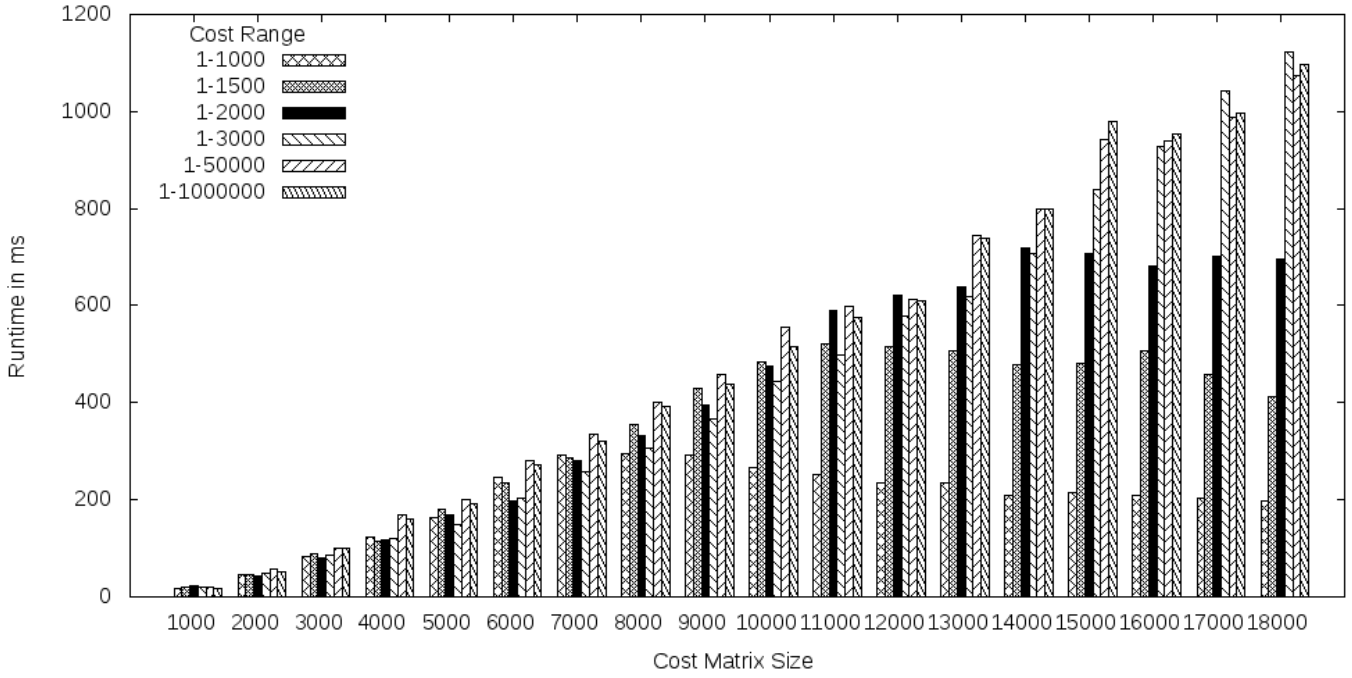

Fig. 5. Runtime of each iteration of augmentation (averaged over all three algorithms).

CFG 1	CFG 2	Size Ratio	VJ-ORG	VJ-CTRL	VJ-IMP	Speedup
bash	BinSlayer	0.27	201	205	125	62%
BinSlayer	bash	3.75	2944	3164	1820	68%
cmaker	BinSlayer	0.11	623	627	264	137%
BinSlayer	cmaker	8.83	13221	13729	8905	51%
cmaker	bash	0.42	3763	3901	3080	24%
bash	cmaker	2.35	54308	58333	43576	29%
gdb	BinSlayer	0.07	1862	1656	588	199%
BinSlayer	gdb	15.02	34020	35468	22127	57%
gdb	bash	0.25	4555	5087	3155	53%
bash	gdb	4.00	151457	160074	114091	37%
gdb	cmaker	0.59	17987	18495	15291	19%
cmaker	gdb	1.70	209562	228467	164442	33%

Fig. 6. Runtimes in milliseconds, where the size ratio is $|V_1|/|V_2|$.

large top-right (deletion) quadrant and a small bottom-left (addition) quadrant. It is notable that while all versions of VJ are faster when $|V_1| < |V_2|$, the benefits to VJ-IMP are more significant. Excluding augmentation, the runtime of each component of each algorithm is almost constant no matter whether $|V_1| < |V_2|$. However, the runtime of augmentation is smaller when $|V_1| < |V_2|$. Since column reduction is faster in VJ-IMP extra benefits follow from $|V_1| < |V_2|$ because column reduction is faster and the cost of augmentation is less dominant. We have found that the number of iterations in augmentation does not depend on $|V_1| < |V_2|$, and so the decreased time in augmentation is entirely a byproduct of an decrease in the runtime of each iteration.

5.3. Component Analysis

Figure 7 shows the time proportion spent in each component of VJ-ORG and VJ-IMP. Column reduction and augmenting row reduction benefit most from the improvements; augmentation is faster with VJ-IMP (in absolute terms).

Augmentation and column reduction take up the majority of the runtime followed by augmenting row reduction, with reduction transfer being almost insignificant. We notice a significant overall speedup with the improved algorithm, but see a huge difference in the percentage of total runtime that the dominant augmentation step takes, hence we can conclude the improvements are most significant over augmenting row reduction and column reduction. Inspection of the raw data revealed column reduction and augmenting row reduction exhibiting between and 2 and 5 fold increase in speed, depending on cost matrix size and range. This dramatic improvement partially stems from the way these steps successively access rows in the cost matrix. Reducing the width of a row in the matrix, as with the new representation, makes it more likely to have data already available in the cache and thus avoid having to access disk storage. The reduction in row size allows more of the relevant portions of the row to be stored in in the data cache (D1) , speeding up accesses as the algorithm accesses rows one after the other. Profiling with valgrind, revealed that the new representation significantly reduced the number of D1 cache misses from approximately 139 million to 48 million, which indicates that the reduction in row width uses the cache more efficiently. Augmentation does not access rows in succession and therefore we observe only a modest performance gain.

We also see an interesting behaviour in the runtimes of augmentation and column reduction, especially at low cost ranges. Column reduction quickly increases as a percentage of total runtime as cost matrix size increases eventually overtaking augmentation. On closer examination, it is apparent that this is the result of a reduction in the growth of the runtime of augmentation instead of a sharp increase in the runtime of column reduction. Furthermore this happens across all cost ranges (but is more visible for smaller ranges). This is surprising as augmentation is merely an implementation of Dijkstra's algorithm. Figure 5 suggests that this stems from an effect in which the growth in runtime of each iteration of augmentation actually tails off as the size of cost matrix increases.

We conjecture that this is because of a statistical property related to the birthday paradox. During initialisation the column reduction step works backward (from the highest index to the lowest), so low index columns have a higher chance of involving a collision and having a lowest element in the same position as another column. For a randomly generated matrix of total size $t \times t$, column index i will have a $(\frac{t}{t+1})^{t-i}$ probability of not having a minimum element in the same row as another column, and thus low i are very likely to be unassigned. Thus a column with a given index is more likely to be unassigned as cost matrix size increases. Consequently not only are low indexed columns more likely to be unassigned, but across all columns this effect will increase disproportionately as matrix size increases. Since Dijkstra's algorithm scans from low indexed columns to high indexed ones, it will find assignments for most of its rows more quickly as cost matrix size increases, even though worse-case complexity remains in $O(n^3)$.

6. Discussion

Serratosa (2014) likewise improves bipartite graph matching, but from the perspective of a refined definition of edit distance. By assuming the properties which include:

1. $0 \leq c_{i,j}$, $0 \leq c_{i,\epsilon}$ and $0 \leq c_{\epsilon,j}$,

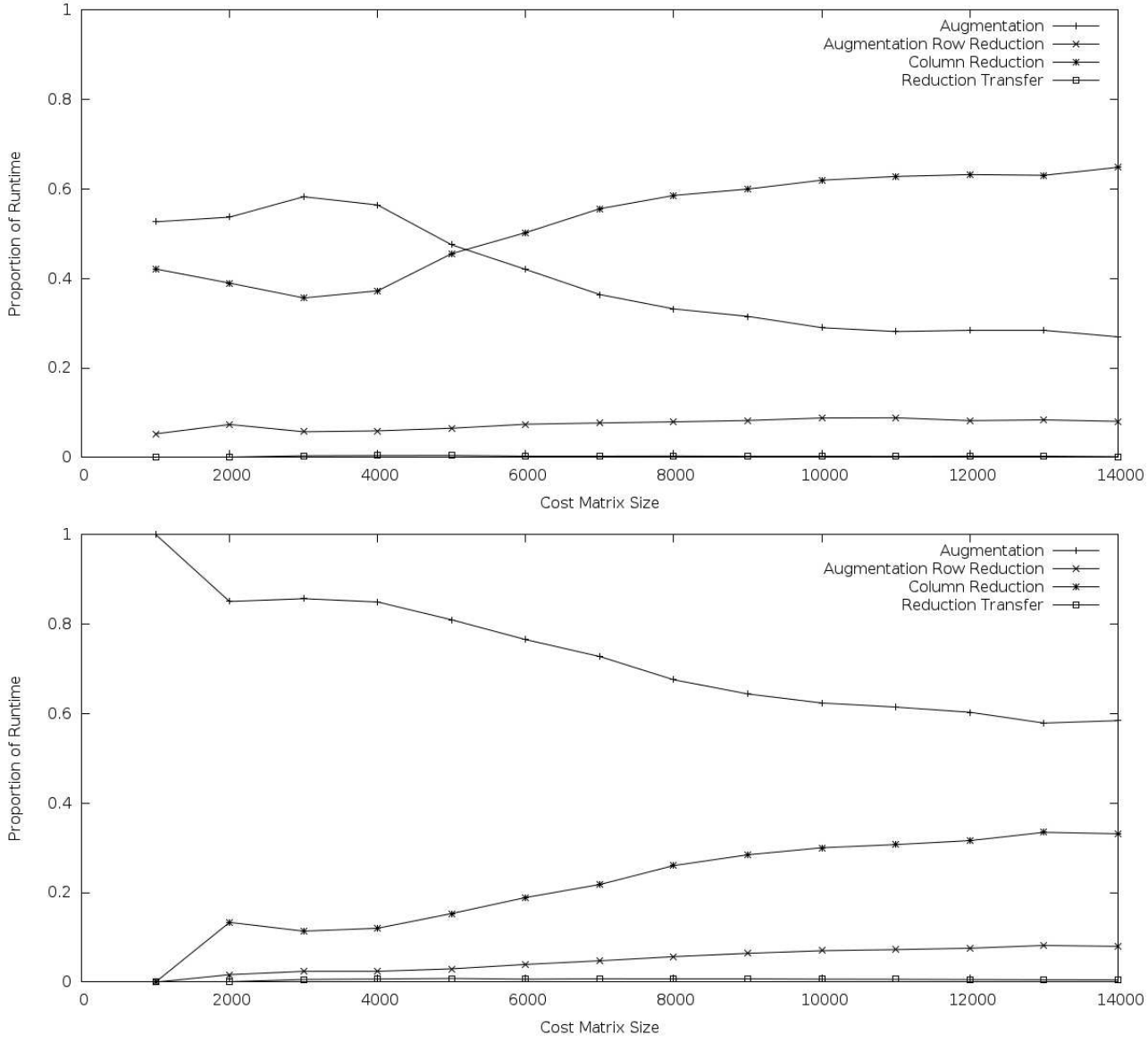

Fig. 7. Normalised runtime of components, for VJ-ORG **(above) and** VJ-IMP **(below), over range 1-500**

2. $c_{i,\epsilon} = c_{\epsilon,j}$,
3. $c_{i,j} \leq c_{i,\epsilon} + c_{\epsilon,j}$,
4. $c_{i,j} = 0$ if $\ell(i) = \ell(j)$, that is, vertices i and j share the same attribute.

it follows that it is not possible for an optimal edit path to include both insertion and deletion operations.This simplifies the calculation of the edit path. It is natural to ask how these assumptions relate to the VJ algorithm, what potential they offer to further improve the representation, and whether any speedups are likely to ensue.

Assumption (1) validates the non-negativity requirement of VJ algorithm. With assumption (2) it is only necessary to store one of the two diagonals, making redundant either the (deletion) column or the (insertion) row. In addition, our proposed representation could be further simplified if the top-left quadrant was symmetric, that is, $c_{i,j} = c_{j,i}$. This would half again the memory requirements but unfortunately is not always a reasonable assumption. Compounding assumption (2), assumption (3) allows us to potentially reduce the number of required calculations. Where assumption (2) allows us to reduce the representation of the cost matrix by only storing either insertions or deletions, assumption (3) actually precludes any solution from containing both edit operations. With regards to assumption (4), while it seems to superficially contradict our proposal in sections 4.2, 4.3 and 4.4 to expedite some calculations by assuming non-zero costs, assumption (4) only relates to the top-left (substitution) quadrant, whereas we only assume non-zero costs for addition or deletion.

An interesting side effect of the representation is that the fixed infinite values in the cost matrix are not directly represented, instead they are introduced on-demand when required (in augmentation). Because of this, we chose to deploy ex-

tended arithmetic in which addition or subtraction of any integer with the symbolic infinite values retained the infinite value. This, and the improvements that skip over infinite values in certain operations, potentially benefit the manipulation of matrices that contain large entries. Extended arithmetic makes it truly impossible to make nonsensical assignments to these infinite values, whereas it was previously assumed that representing infinite values with sufficiently large numbers would suffice to prevent this.

Profiling suggests that cache misses determine the performance of the VJ algorithm. This suggests that VJ needs to revisited, with a view to reformulation its steps so as to minimise the number of cache misses. Such engineering could enable VJ to be scaled to significantly larger problems.

Unlike the Hungarian method which relies on an integral duality theorem that does not hold for rational cost matrices in general (Kuhn, 1955), there are no fundamental problems that similarly constrain the VJ algorithm. However, there are inherent problems instantiating the VJ representation with floating point numbers, requiring the introduction of a tolerance to deal with loss of precision. The introduction of the new representation can mitigate this by avoiding a large number of unnecessary floating point operations.

7. Conclusions

We have examined the VJ algorithm and studied improvements for approximating GED. We have shown that our improved algorithm is uniformly faster than its unimproved counterparts, both across randomly generated matrices and data sets that arise in call-graph comparison. The speedups, which are almost 200% in one case, suggest the that refinements are truly worthwhile. Moreover, the improved version has a smaller memory footprint, and incurs no loss of accuracy whatsoever. Finally, we have also documented and explained an anomaly in the runtime of the Dijkstra's shortest path search component of VJ. Future work will, among other things, empirically investigate how the relative sizes of the two graphs under comparison effect the overall runtime, and also explore the prospects for parallelisation (Balasn et al., 1991).

Acknowledgements

We thank Earl Barr and David Clark for discussions on malware matching that motivated this work, and the reviewers of this paper who had several important insights into our work, and Laetitia Pelacchi for her support. Finally, we are indebted to Francesc Serratosa for helpful and thought-provoking discussions. This work was supported, in part, by grant EP/K031929/1 funded by GCHQ in association with EPSRC.

References

E. Balasn, D. Miller, J Pekny, and P. Toth. A parallel shortest augmenting path algorithm for the assignment problem. *JACM*, 38(4):985–1004, 1991.

M. Bourquin, A. King, and E. Robbins. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of Program Protection and Reverse Engineering Workshop*. ACM, 2013.

H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

R. E. Burkard and E. Cela. *Linear Assignment Problems and Extensions*. Springer, 1999.

D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

M. A. Eshera and K.-S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man and Cybernetics*, (3):398–408, 1984.

X. Gao, B. Xiao, D. Tao, and X. Li. A Survey of Graph Edit Distance. *Pattern Analysis and Applications*, 13(1):113–129, 2010.

W. Jones, A. Chawdhary, and A. King. Revisiting Volgenant-Jonker for Approximating Graph Edit Distance. In C.-L. Liu, B. Luo, W. G. Kropatsch, and J. Cheng, editors, *Graph-based Representations in Pattern Recognition*, pages 98–107. Springer, 2015.

R. Jonker and A. Volgenant. A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. *Computing*, 38(4):325–340, 1987.

H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

L. Livi and A. Rizzi. The graph matching problem. *Pattern Analysis and Applications*, 16(3):253–283, 2013.

J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics*, 5(1):32–38, 1957.

R. Myers, R. C. Wison, and E. R. Hancock. Bayesian graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(6):628–635, 2000.

K. Riesen and H. Bunke. Approximate Graph Edit Distance computation by means of Bipartite Graph Matching. *Image and Vision Computing*, 27(7): 950–959, 2009.

K. Riesen, M. Neuhaus, and H. Bunke. Bipartite Graph Matching for Computing the Edit Distance of Graphs. In *Graph-Based Representations in Pattern Recognition*, pages 1–12. Springer, 2007.

A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man and Cybernetics*, (3):353–362, 1983.

F. Serratosa. Fast computation of bipartite graph matching. *Pattern Recognition Letters*, 45:244–250, 2014.

F. Serratosa and X. Cortés. Edit Distance Computed by Fast Bipartite Graph Matching. In *Structural, Syntactic, and Statistical Pattern Recognition*, pages 253–262. Springer, 2014.

Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing Stars: On Approximating Graph Edit Distance. *VLDB Endowment*, 2(1):25–36, 2009.